A short unit to introduce multi-threaded programming*

David P. Bunde
Knox College
dbunde@knox.edu

Abstract We argue for the inclusion of concurrent programming in core courses of the computer science major and present a brief unit that quickly introduces some key concepts.

1 Introduction

Computing has long benefited from Moore's law, often stated as claiming that microprocessor performance doubles roughly every 18 months. This exponential growth has been achieved since the 1970s through a combination of exponential growth in both the number of transistors on a chip and the speed at which chips are clocked. Unfortunately, clock speed seems unlikely to increase further because processor power consumption grows superlinearly with clock speed and heat dissipation is reaching its limits. The number of transistors on a processor continues to grow, but traditional ways of using them for performance, such as instruction-level parallelism, are also reaching their limits. Together, these factors seem to require that additional performance come from adding processing elements rather than increasing the performance of an individual processing element. Indeed, this direction is being pursued by microprocessor manufacturers, all of whom are designing and building processors with multiple cores per chip. A typical computer purchased today has two cores, with larger numbers available in high-end systems.

How does the introduction of multiple cores affect programmers and, by extension, computer science educators? The obvious way to use multiple cores is simply to assign each a different program. This idea works as long as the user runs enough programs to use all the cores, but it has definite limits. Since an Intel engineer recently wrote "developers should start thinking about tens, hundreds, and thousands of cores" in a corporate blog [8], it is clear that we will want multiple cores working on a single application. The imminent emergence of truly large numbers of cores per processor (so called *manycore* systems) is now conventional wisdom for many computer science researchers (cf. [3]).

Although manycore processors do not yet exist and no one knows exactly how they will be programmed, we believe that exposing students to the general principles of concurrent programming can (and should) begin immediately. This paper describes an effort to create a short unit introducing concurrent programming that fits within an existing course. Our unit took the form of a lab activity and followup homework assignment on multi-threaded programming. Importantly, the course was unchanged except for this lab, the homework, and about half a period of followup discussion

*©CCSC, 2009. This is the author's version of the work. It is posted here by permission of CCSC for your personal use. Not for redistribution.

during lecture. With this minimal change, we were able to quickly introduce the following concepts: the fork/join paradigm, race conditions, parallel speedup, and load balance. Multi-threaded programming seems to be a particularly appropriate way to address the manycore challenge since cores sharing a chip are more likely to support a shared memory space than larger-scale systems. In addition, as we show, it is relatively easy to quickly present threads to students.

The general problem used in our assignments is hinted at in the Java documentation for the `Thread` class. What is novel about our unit is the ease of fitting it into an already-crowded curriculum. Synchronization and race conditions are often taught in operating systems (OS) courses as part of the concurrency core topic (OS3) specified by the CC2001 curriculum [20] and we simply focused part of that topic on multi-threaded programming.

This course change did not require any special computing resources. At the time, our lab computers were dual-processor AMD Athlon MP 1900+s. These older machines (released in 2001!) were among the first multiprocessor desktops, but our assignments also work on run-of-the-mill newer machines, which are typically dual-core. Furthermore, the assignments will get more interesting over time as new processors have more cores and allow higher levels of concurrency.

Another virtue of our approach is that it exposes students to concurrent programming in a required course. Most courses teaching this topic are specialized electives [27], which limits the number of students who see this material.

Despite the positive attributes of our concurrency unit, let us make it clear that we see much room for improvement. Our assignments solve a toy problem. This exposes the underlying issues, but does not provide as much motivation as a more realistic problem. The very briefness of the unit also lessens its effectiveness since students do not have much chance to practice or develop intuition. It would undoubtedly be more effective to discuss concurrency throughout the curriculum as advocated by others (cf. [7, 26]). Since curricula change slowly, however, we assert the value of quick "band-aid" solutions as well as more thorough approaches.

In the rest of this paper, we summarize previous work on teaching concurrent programming, discuss relevant parts of the Java API, give our sample unit, and then discuss possible extensions.

2 Related Work

We are far from the first to advocate a greater focus on concurrent programming in computer science education. Nearly fifteen years ago, there was even a workshop exclusively devoted to this topic [25, 21]. What has changed in the intervening time is the advent of multicore machines.

Our sample unit uses Java because we liked its interface for threading and synchronization. (The synchronization commands were surprisingly tricky in early versions of Java [13, 15], but these problems have been fixed in version 1.5.) We do not view language selection as crucial for multi-threaded programming provided the API is clean enough to not obscure the underlying concepts. Particularly well suited are languages with built-in concurrency support. Goldwasser and Letscher [10] discuss a Python-based CS 1 course in which students write a multi-threaded chat server as part of a module on networking. This ambitious project is facilitated by the ease of creating threads in Python. Sheehan [30] describes the benefits of teaching operating systems in Ruby, which implements user-level threads and provides concurrency control features such as mutexes. Ada has also been used since it has built-in task creation and synchronization primitives [4, 17, 29].

A more traditional introduction to multi-threaded programming uses C, a choice inherited from its use in the OS course, where interprocess communication (IPC) is a typical topic. Threads and

synchronization primitives are not built into C, but POSIX-standard implementations of threading (pthreads) and semaphores (`sem_init`, `sem_wait`, `sem_post`, ...) are widely available. To further simplify their use, libraries [5, 16] and a simulator [22] have been developed to provide nicer interfaces and debugging support. Most of the rest of the course in which our unit was taught used C since other course goals were to expose students to pointers, manual memory management, and linking. This led a colleague to adapt our unit to C for a subsequent iteration of the course.

Another approach to teaching concurrent programming is to use a special-purpose language. Exemplifying this trend is the work of Jacobsen and Jadud [18], who use occam-π to program robots. When using occam-π, the programmer defines a system of processes[1] joined by unidirectional communication channels. The compiler and runtime system distribute these processes among processors/cores and handle synchronization of the communication channels. Both of these services hide complexity from the programmer. The language also includes a construct for the programmer to indicate the independence of statements in a code block, allowing the system to run them in parallel. A course based on this language has been taught to undergraduates and as a workshop for experienced software professionals [19]. Other courses using special-purpose languages are described by Hartley [12] and McDonald [24].

Other library options are MPI and OpenMP, typically used for scientific computing. Apon et al. [2] describe several parallel programming courses using these libraries, with an emphasis on MPI. Pan [28] describes a course teaching both MPI and OpenMP rather than focusing on only one. All of these courses used a cluster as the parallel system, which differs from our focus on multi-threaded programming.

Shene [32] describes a course similar to ours, but with a much greater emphasis on multi-threaded programming. (All 4 assignments during the quarter involved threads and IPC, followed by writing a thread system as the final project.) His first assignment is a fork-join program with no other synchronization such as matrix multiplication (one thread per element) or quicksort (one thread per partition). These are similar toy problems to the one we discuss in this paper, but we add explicit consideration of load balancing and more synchronization. Shene also wrote a tutorial [31], which is a good resource on these concepts and looks at some other problems.

Jackson [17] describes a 5-lecture "mini-course" to teach concurrency. The first of these is on processes, part of any OS course, but the other four cover synchronization and race conditions as well as topics beyond the scope of our unit.

Ernst and Stevenson [7] describe their work to introduce concurrency throughout their curriculum. They have students use threads to solve embarrassingly parallel problems in CS 1. Simple synchronization requirements are introduced in CS 2 using problems like ray tracing and more complicated synchronization is discussed in Algorithms, where multi-dimensional dynamic programming is used as a motivating problem.

Also related are some textbooks [14, 23] that teach concurrency in Java, though using versions before 1.5 without high-level features like semaphores. A recent trade book is Goetz et al. [9].

3 Threads in Java

Now we give a brief overview of some relevant parts of the Java API. A key class for writing multi-threaded programs is `java.lang.Thread`, which encapsulates a single thread of execution.

[1] We retain the terminology of Jacobsen and Jadud [18], but the reader should note the difference between OS processes and these processes, which the occam-π runtime system manages.

Once a `Thread` is created, it starts execution when its `start()` method is called. This method marks the `Thread` object ready and returns, allowing the caller to continue execution. The started `Thread` executes its `run()` method. If the caller (or some other piece of code) wants to wait for this `Thread` to finish, it calls the `Thread`'s method `join()`, which blocks until `run()` returns.

Since it is up to the programmer to specify what a `Thread` does, by default `run()` does nothing and returns immediately. There are two ways to specify work for a `Thread` to perform. Most directly, the programmer can create a class extending `Thread` and override `run()`. Alternately, the programmer can create a class implementing the `Runnable` interface, which has a single method `run()`. Passing a `Runnable` object to the constructor of `Thread` causes that `Thread`'s `run()` method to invoke the `run()` method of the `Runnable` object. In our course, we used the second approach since it seemed conceptually simpler to pass work to a thread than override part of a complex class.

Creating `Thread` objects allows programmers to easily create a program with multiple threads. It is nearly always necessary for these threads to communicate. On one hand, this is simple since threads share an address space and can share variables directly. On the other hand, it is necessary to prevent one thread from accessing data during an update performed by another thread. Again, Java provides a simple way to do this. The keyword `synchronized` specifies that the lock attached to a particular object must be held during a code block. For example, the code

```
synchronized(obj) {
  ...
}
```

acquires the lock attached to `obj`, runs the block, and then releases the lock. The object `obj` can be a member of any class. A thread trying to acquire a lock that is already held will block. For convenience, `synchronized` can also modify a method signature, as in

```
public synchronized void method() {
  ...
}
```

which is equivalent to placing the method body inside a `synchronized` block that locks on `this`. Java also has more general synchronization primitives such as semaphores in `java.util.concurrent`, but these are not used in our unit.

4 Sample Unit

Now we discuss our sample unit. It was covered midway through a course based on "Operating systems and networking" in the CC2001 small department curriculum [20]. Our course meets for lecture three times a week and for lab once a week. Each meeting lasts for 70 minutes so we achieve the contact hours of a 15-week semester in only 10 weeks. (For more details about our curriculum, see Dooley [6].) Prior to the lab below, we had discussed mutexes in the context of classic IPC problems (Dining Philosophers and Readers and Writers), generally following the treatment in Tanenbaum [33], but the students had no previous exposure to multi-threaded programming.

4.1 Lab exercise

Our coverage of multi-threaded programming began with a lab in which the students followed a handout that provided background and guided them through the assignment. During lab, the in-

structor wandered between students answering questions. Students are not graded on lab exercises, but attend and participate anyway because lab exercises often relate to homework.

At the beginning of lab, the students examine a simple serial program to count the number of primes in the range 2–2,000,000. This program uses a helper function `isPrime()` to test each odd integer in the range and then prints the number of primes. The `isPrime()` function takes an integer n and tests whether any integer from 2 to $\sqrt{n}$ divides it. The students are given this version to remind them of prime numbers and let them examine the program without any threading code.

Next, the students examine a naive attempt to parallelize this program, the bulk of which appears in Figure 1. This program uses an inner class `PrimeFinder` for the task of counting primes in some range. The program creates one thread for the range 3–1,000,000 and one for 1,000,001–2,000,000. As primes are found, the shared variable `pCount` is incremented.

The given code has two fatal bugs. First of all, `main()` does not wait for the threads to finish before printing the result. Second, access to `pCount` is not synchronized, resulting in a race condition. The students are told to run the initial program and then fix the first bug. This bug gives them a reason to examine the code and learn the parts of a fork/join program in Java. The resulting program still slightly undercounts the number of primes and they are asked to find the reason. (The handout has a strategic page break to prevent them from accidentally reading ahead.) This bug is meant to be somewhat puzzling; it should be findable based on the coverage of mutexes in class, but is hidden because it occurs when a thread is interrupted in the middle of a line of Java code. The challenge of finding this bug makes the idea of a race condition more memorable and illustrates how hard they can be to find.

The obvious fix for the race condition is to put the line `pCount++` in a `synchronized` block. The students are told to do so and then observe that the program works correctly.

The rest of the lab is concerned with performance. The handout has them use the Unix command `time` to compute the program's speedup, a concept the students had not seen previously.

To improve performance, the students are first directed to reduce the contention on the shared variable `pCount`. Instead of having the threads acquire the lock each time a prime is found, a more efficient solution is for each thread to keep a private count of the primes it finds and add this number to the global sum after examining its entire range. This change greatly reduces the number of times the lock is acquired. Although the performance impact turns out to be minimal (the critical section is short and the threads encounter primes at different enough times to minimize contention), the idea of making private copies to reduce contention is an important one for the students to see.

The resulting program has a speedup of 1.6, hardly impressive since the problem is so easy to split into completely independent components. The problem is that the load is unbalanced; on average, it takes longer to test larger numbers for primality. The lab concludes by asking the students to identify and fix this problem themselves. They were given the following hint: "What happens when you set the two parts to find primes within the ranges 1–1,100,000 and 1,100,001–2,000,000 instead of splitting the test region evenly?".

We discussed this lab in the following lecture, reviewing fork/join programs, race conditions, speedup, and load balance. We specifically talked about achieving better load balance in the hunt for primes. Most of the students, perhaps led astray by the handout's hint, balanced the load by manipulating the size of each range. We talked about how this approach is not very useful without a way to estimate how long each part takes (including a brief aside about the possibility of estimating the load in our problem using the distribution of prime numbers). One student split the range into more than two parts, each with its own thread. We talked about how this approach

```
public class ThreadedPrimes {
  public static int pCount;

  static class PrimeFinder implements Runnable {
    private long from;
    private long to;

    //constructor & isPrime omitted...

    public void run() {
      long nextCand = from;

      while(nextCand < to) {
        if(isPrime(nextCand))
          pCount++;

        nextCand += 2;
      }
    }
  }

  public static void main(String args[]) throws InterruptedException {
    pCount = 1;   //(already found 2)

    PrimeFinder p1 = new PrimeFinder(3,1000000);
    Thread t1 = new Thread(p1);
    PrimeFinder p2 = new PrimeFinder(1000001, 2000000);
    Thread t2 = new Thread(p2);

    t1.start();
    t2.start();

    System.out.println(pCount + " primes");
  }
}
```

Figure 1: Initial (broken) multi-threaded program

creates a more balanced load since a core that finishes more quickly simply takes another part. Following their idea to its logical conclusion, this student had discovered that poor performance results from using a very large number of threads, which led to a discussion of thread creation overhead and the possibility of using a work pool rather than a separate thread for each part. We also talked about what seems to be the most elegant solution: one part looks for primes of the form $4k+1$ over the entire range and the other looks for those of the form $4k+3$. Each part contains numbers of equivalent size and the load balances nicely.

4.2 Homework assignment

Following this discussion, the students did a related homework assignment. The basis for this homework is a faster serial version of the prime counting program. Instead of `isPrime()` trying to divide its argument n by each number from 2 to $\sqrt{n}$, the new version only tries to divide n by the primes in this range. To do so, the program must keep a list of the primes identified so far, but the change saves many division operations. The new implementation of `isPrime()` appears in Figure 2. The sequential program with this improvement is faster than the best concurrent program developed in lab, demonstrating a variation of the saying "premature optimization is the root of all evil" (attributed to both Knuth and Hoare). In our case, we made a slow program multi-threaded rather than finding the faster version and then multi-threading it. (Hard to do in practice, but a point worth making in any case.)

It is not as clear how to use multiple threads with this version of `isPrime()` because calls to `isPrime()` on larger numbers now depend on the result of earlier calls. Since it was harder, the assignment was given in two parts. In the first part, students are told to parallelize the code using the approach that ultimately works: first, the program sequentially finds the primes up to $\sqrt{2,000,000}$, which allows `isPrime()` to work on all the values it is called on. Then the program uses a pair of threads to count the other primes. Small primes are stored in an `ArrayList` and the others are counted using a private variable for each thread. Using an `ArrayList` rather than its synchronized cousin `Vector` is crucial to avoid synchronization overhead and works even with multiple threads because its contents are not modified once simultaneous accesses begin.

For the assignment's second part, students are asked about parallelizing this program in the same way as the initial prime-counting program; use a `synchronized` block to control access to

```
public static boolean isPrime(long num) {
  long limit = (long) Math.sqrt(num);

  long val;
  int i = 1; //num is odd so skip 2
  while((i<primes.size()) && ((val=primes.get(i)) <= limit)) {
    if(num % val == 0)
      return false;
    i++;
  }

  return true;
}
```

Figure 2: Using a sieve; `primes` is `ArrayList` of known primes (in increasing order)

the `ArrayList` and have two threads identify primes of the form $4k + 1$ and $4k + 3$ respectively. Specifically, they are told that this version suffers from overhead and a race condition, then asked to identify the source of each. (The overhead is contention entering the `synchronized` block; the critical section is small, but it is inside the main loop of `isPrime()`. The race condition involves the order in which primes are found and added to the list; `isPrime()` assumes that the primes are sorted (and is slower without this assumption).)

4.3 Evaluation

Overall, we were pleased with the outcome of our unit. The students generally did well on the homework assignment, earning a higher average than for homeworks overall. The only conceptual problem that appeared on the first part was that 2 students (out of 8) modified `isPrime()` so that on small numbers, it tested numbers 3 through $\sqrt{n}$ as possible divisors in addition to trying the already-identified primes. This error makes the program inefficient, but not incorrect. We believe it was caused by lack of familiarity with prime numbers rather than a misunderstanding of concurrency.

There were more errors on the second part of the homework. The main cause seems to be confusion about what the problem required, however, because the description of the naive threaded version was rather vague. In retrospect, the assignment should have included code for this version, which had been written while creating the assignment.

The students seemed to appreciate this unit. They were asked to comment on it in their course evaluations and the only comment received was "I enjoyed the multithreading part of the course– in fact, I think it could have been even more challenging." Several students have asked for more coverage of concurrent programming in person or via email.

Based on the student feedback and how well they did on the assignment, it probably should have been harder. Since the students had never done concurrent programming, the assignment was designed to be fairly straightforward and brief (they were given only 3 days). The next iteration of this assignment will require them to do more of the design for the multi-threaded program.

5 Discussion

We have described the need to expose students to concurrent programming and presented a short unit that attempts to do so. This unit consists of a single lab period, a short homework, and a brief in-class discussion. This minimal commitment makes the unit easy to add to an existing course. Materials for this unit, including handouts and given code, are available at `http://faculty.knox.edu/dbunde/teaching/threadIntro`.

Although we believe this unit is a start, there is plenty of room for improvement. Ideally, concurrency will be presented more thoroughly and more often than is possible with a single small unit. Even if the coverage remains brief, it would be preferable to use a problem more interesting than counting prime numbers. Ideal would be to use several problems that vary in how well they load balance since linear speedup is often not a realistic goal. Programs with a longer serial section would also allow introduction of Amdahl's law. We hope others are inspired to create materials and assignments. Steve Heller has created a wiki [27] for use in getting and sharing ideas.

Another question that arises is how to scale up this type of assignment to a higher level of concurrency. Processors in the future may have hundreds of cores, but typical machines today

have only two. Many scalability issues are hidden with such a low level of concurrency, making dual-core machines unsatisfying as a testbed. One solution is to use a server-class machine. A more inexpensive solution (with higher latency) would be to use a small cluster of machines, either lab machines or an Microwulf cluster [1]. Another possibility is being explored by the RAMP project [11], which aims to build simple manycore processors in FPGAs.

References

[1] J.C. Adams and T.H. Brom. Microwulf: A beowulf cluster for every desk. In *Proc. 39th SIGCSE Technical Symp. Computer Science Education*, pages 121–125, 2008.

[2] A. Apon, J. Mache, R. Buyya, and H. Jin. Cluster computing in the classroom and integration with Computing Curricula 2001. *IEEE Trans. on Education*, 47(2):188–195, 2004.

[3] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and K.A. Yelick. The landscape of parallel computing research: A view from Berkeley. Tech Report UCB/EECS-2006-183, EECS Department, UC Berkeley, Dec 2006. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

[4] M. Ben-Ari. Using inheritance to implement concurrency. In *Proc. 27th SIGCSE Technical Symp. Computer Science Education*, pages 180–184, 1996.

[5] S. Carr, J. Mayo, and C.-K. Shene. ThreadMentor: A pedagogical tool for multithreaded programming. *ACM J. Educational Resources in Computing*, 3(1), 2003.

[6] J.F. Dooley. Moving to CC2001 at a small college. In *Proc. 9th Ann. SIGCSE Conf. Innovation and Technology in Computer Science Education*, pages 196–198, 2004.

[7] D.J. Ernst and D.E. Stevenson. Concurrent CS: Preparing students for a multicore world. In *Proc. 13th Ann. SIGCSE Conf. Innovation and Technology in Computer Science Education*, pages 230–234, 2008.

[8] A. Ghuloum. Unwelcome advice. Blog post. `http://blogs.intel.com/research/2008/06/unwelcome_advice.php`, June 30 2008.

[9] B. Goetz, T. Peieris, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java concurrency in practice*. Addison-Wesley, 2006.

[10] M.H. Goldwasser and D. Letscher. Introducing network programming into a CS 1 course. In *Proc. 12th Ann. SIGCSE Conf. Innovation and Technology in Computer Science Education*, pages 19–22, 2007.

[11] RAMP Group. Research accelerator for multiple processors. `http://ramp.eecs.berkeley.edu/`.

[12] S.J. Hartley. Experience with the language SR in an undergraduate operating systems course. In *Proc. 23rd SIGCSE Technical Symp. Computer Science Education*, pages 176–180, 1992.

[13] S.J. Hartley. Alfonse, your java is ready! In *Proc. 29th SIGCSE Technical Symp. Computer Science Education*, pages 247–251, 1998.

[14] S.J. Hartley. *Concurrent programming: The Java programming language*. Oxford University Press, 1998.

[15] S.J. Hartley. Alfonse, wait here for my signal! In *Proc. 30th SIGCSE Technical Symp. Computer Science Education*, pages 58–62, 1999.

[16] C.W. Higginbotham and R. Morelli. A system for teaching concurrent programming. In *Proc. 22nd SIGCSE Technical Symp. Computer Science Education*, pages 309–316, 1991.

[17] D. Jackson. A mini-course on concurrency. In *Proc. 22nd SIGCSE Technical Symp. Computer Science Education*, pages 92–96, 1991.

[18] C.L. Jacobsen and M.C. Jadud. Towards concrete concurrency: occam-pi on the LEGO Mindstorms. In *Proc. 36th SIGCSE Technical Symp. Computer Science Education*, pages 431–435, 2005.

[19] M.C. Jadud, J. Simpson, and C.L. Jacobsen. Patterns for programming in parallel, pedagogically. In *Proc. 39th SIGCSE Technical Symp. Computer Science Education*, pages 231–235, 2008.

[20] Joint task force on computing curricula, IEEE Computer Society and Association for Computing Machinery. *Computing Curriculum 2001: Report of the joint task force on computing curricula*, 2001. `http://www.sigcse.org/cc2001/`.

[21] D. Kotz, editor. *Proc. 2nd Forum on Parallel Computing Curricula*, 1997. `http://www.cs.dartmouth.edu/FPCC/papers/`.

[22] B.L. Kurtz, H. Cai, C. Plock, and X. Chen. A concurrency simulator designed for sophomore-level instruction. In *Proc. 29th SIGCSE Technical Symp. Computer Science Education*, pages 237–241, 1998.

[23] D. Lea. *Concurrent programming in Java: Design principles and pattern*. Prentice Hall, 2nd edition, 2000.

[24] C. McDonald. Teaching concurrency with Joyce and Linda. In *Proc. 23rd SIGCSE Technical Symp. Computer Science Education*, pages 46–52, 1992.

[25] P.T. Metaxas and M. Merzbacher, editors. *Proc. Forum on Parallel Computing Curricula*, 1995. `http://www.wellesley.edu/CS/forum/pcc.html`.

[26] C.H. Nevison. Parallel computing in the undergraduate curriculum. *Computer*, 28(12):51–56, 1995.

[27] OpenSPARC. *Sharing Teaching Material for Concurrent Computing*, Wiki viewed Aug 2008. `http://wiki.opensparc.net/bin/view.pl/CourseMaterial/ConcurrentComputing`.

[28] L. Pan. An innovative course in parallel computing. *J. STEM Education Innovations and Research*, 4, 2003.

[29] Y. Persky and M. Ben-Ari. Re-engineering a concurrency simulator. In *Proc. 3rd Ann. SIGCSE Conf. Innovation and Technology in Computer Science Education*, pages 185–188, 1998.

[30] R. Sheehan. Teaching operating systems with Ruby. In *Proc. 12th Ann. SIGCSE Conf. Innovation and Technology in Computer Science Education*, pages 38–42, 2007.

[31] C.-K. Shene. Multithreaded programming with ThreadMentor: A tutorial. `http://www.cs.mtu.edu/~shene/NSF-3/e-Book/`.

[32] C.-K. Shene. Multithreaded programming in an introduction to operating systems course. In *Proc. 29th SIGCSE Technical Symp. Computer Science Education*, pages 242–246, 1998.

[33] A.S. Tanenbaum. *Modern Operating Systems*. Pearson Education, 3rd edition, 2008.